“须知佛法在求渡世,武功在求杀生,两者背道而驰,相互克制。只有慈悲念越盛,武功绝技才能练得越多,但修为上到了如此境界的高僧,却又不屑去学各种厉害的杀人法门了。”--扫地僧

后端技术栈 - 关系数据库篇

[开发环境 5](#_Toc33450353)

[基础原理 5](#_Toc33450354)

[Mysql 体系结构 6](#_Toc33450355)

[Innodb 存储引擎 8](#_Toc33450356)

[缓冲池 8](#_Toc33450357)

[插入缓冲 11](#_Toc33450358)

[Change Buffer 12](#_Toc33450359)

[两次写 14](#_Toc33450360)

[自适应哈希索引 15](#_Toc33450361)

[刷新临接页 16](#_Toc33450362)

[引擎开启与关闭 16](#_Toc33450363)

[内存堆 17](#_Toc33450364)

[重做日志缓冲 17](#_Toc33450365)

[Master Thread 18](#_Toc33450366)

[IO Thread 20](#_Toc33450367)

[Purge Thread 20](#_Toc33450368)

[Page Cleaner Thread 21](#_Toc33450369)

[Checkpoint 21](#_Toc33450370)

[文件系统 22](#_Toc33450371)

[日志文件 22](#_Toc33450372)

[错误日志 22](#_Toc33450373)

[二进制日志 23](#_Toc33450374)

[查询日志 25](#_Toc33450375)

[慢查询日志 25](#_Toc33450376)

[表结构文件 27](#_Toc33450377)

[InnoDB 存储引擎文件 27](#_Toc33450378)

[表空间文件 27](#_Toc33450379)

[共享表空间 28](#_Toc33450380)

[重做日志文件 29](#_Toc33450381)

[Undo log 32](#_Toc33450382)

[索引 34](#_Toc33450383)

[B+树索引 34](#_Toc33450384)

[联合索引 35](#_Toc33450385)

[覆盖索引 35](#_Toc33450386)

[优化器忽略索引 35](#_Toc33450387)

[MMR 35](#_Toc33450388)

[ICP 35](#_Toc33450389)

[锁 35](#_Toc33450390)

[事务 35](#_Toc33450391)

[性能优化 36](#_Toc33450392)

[池化技术 36](#_Toc33450393)

[读写分离 38](#_Toc33450394)

[主从复制部署 39](#_Toc33450395)

[一主多从 39](#_Toc33450396)

[主从互备 39](#_Toc33450397)

[主从复制关键问题 39](#_Toc33450398)

[主从复制参数 39](#_Toc33450399)

[半同步复制 39](#_Toc33450400)

[实战篇 40](#_Toc33450401)

[索引 40](#_Toc33450402)

[Join 操作 40](#_Toc33450403)

[Order by 原理 42](#_Toc33450404)

[全字段排序 42](#_Toc33450405)

[raiwd排序 43](#_Toc33450406)

[全内存 or 使用外部文件 43](#_Toc33450407)

[不需要排序的 order by 43](#_Toc33450408)

[临时表 44](#_Toc33450409)

[Union 执行过程 45](#_Toc33450410)

[Group by 执行过程 46](#_Toc33450411)

[Insert…select 语句 47](#_Toc33450412)

[再论rowid 50](#_Toc33450413)

[Between and VS or VS in VS >= VS exist 50](#_Toc33450414)

[SQL 语句优化小结 50](#_Toc33450415)

[Explain 详解 54](#_Toc33450416)

### 开发环境

Mysql 目前有三个主流版本：

Oracle Mysql，官方版本。

MariaDB，提供更多对服务器的扩展，同时支持更多引擎。

Percona Server, 与官方版本向后兼容的替代品，“抢先版”Mysql。增加了允许用户更紧密查看服务器内部信息的途径（show status命令、information\_schema 中的各类表、更详细的的慢查询日志），增强了InnoDB引擎的可预测性和稳定性。可以和官方版本无缝切换。

本文使用的是官方版本。

Mysql 5.7.26

innodb\_version 5.7.26

OS ubuntu 16.04

### 基础原理

[Mysql 体系结构](#_Mysql_体系结构)

[InnoDB 存储引擎](#_Innodb_存储引擎)

[文件](#_文件系统)

表

索引

锁

事务

#### Mysql 体系结构



图-1 Mysql 体系结构

由图-1可知Mysql框架主要包括连接组件、管理组件、查询接口、解析器、优化器、缓存组件、存储引擎以及底层的文件系统。

Connectors: 接口API，用于不同语言的客户端与服务器进行通信交互。

Connection Pool：连接池组件，执行身份验证，线程重用，连接限制，检查内存，数据缓存。

通过下面命令查看连接以及连接超时时间：

1. show processlist;
2. show **full** processlist;
3. mysql> show variables like 'wait\_timeout';
4. +---------------+-------+
5. | Variable\_name | Value |
6. +---------------+-------+
7. | wait\_timeout  | 28800 | // 默认连接时长8小时
8. +---------------+-------+

Cache 和 Buffer：缓存组件，对于查询请求，Mysql 会先查询缓存，缓存命中则直接返回，否则执行后续的查询(访问磁盘文件)。

一般情况下不建议使用查询缓存，因为查询缓存的失效非常频繁，只要有对表的更新操作，该表上的所有缓存都会失效，这往往导致极低的缓存命中率，除非业务表是静态表，极少更新。

涉及参数 query\_cache\_type

1. mysql> show variables like 'query\_cache\_type';
2. +------------------+-------+
3. | Variable\_name    | Value |
4. +------------------+-------+
5. | query\_cache\_type | **OFF**   |
6. +------------------+-------+
7. 可以将该值设置为DEMAND，这样默认SQL不使用查询缓存
8. 对于确定需要使用查询缓存的语句，使用  SQL\_CACHE显式指定：
9. **select** SQL\_CACHE \* **from** table\_t **where** col=xxx；

MySQL 8.0 版本直接将查询缓存的整块功能删掉了。

Management Services & Uttilities: 管理服务和工具组件，执行数据的备份和恢复，安全，管理，配置、迁移、元数据等等。

SQL Interface：SQL 接口组件，进行 DML、DDL，存储过程、视图、触发器等操作和管理；用户通过 SQL 命令来与服务器交互。

Parser：查询分析器组件，主要执行语法分析，词法分析，同时还会执行判断语句涉及的表是否存在，表结构是否正确等操作。

Optimizer：优化器组件，生成执行计划。比如表中涉及多个索引的时候，决定使用哪个索引；多表关联（join）的时候，决定各个表的连接顺序。

Pluggable Storage Engines: 插件式存储引擎， 负责数据的存储以及提取。

可以通过SHOW ENGINES命令查看各个存储引擎信息。

#### Innodb 存储引擎

Mysql 本身支持多种存储引擎，比如MyISAM、CSV、MEMORY、BLACKHOLE、InnoDB等，每个存储引擎都有各自的特点以及适用场景。本文仅关注InnoDB引擎。

InnoDB 存储引擎支持事务，主要面向在线事务处理(OLTP)的应用，支持行锁、支持外键、支持一致性非锁定读。具有插入缓冲、两次写、自适应哈希索引、异步IO、刷新邻接页等特性。

InnoDB 引擎包括多个内存块，同时运行多个后台线程。这些内存块负责维护所有进程/线程所需的内部数据结构（内存堆），缓存磁盘上的数据（缓冲池），充当日志缓冲区（重做日志缓冲）等。后台线程负责刷新内存池中的数据，并将更新的数据文件刷新到磁盘文件，以及进行异常恢复。这些线程包括Master Thread、IO Thread、Purge Thread、Page Cleaner Thread。

##### 缓冲池

缓冲池本身是一块固定大小的内存区域，其中缓存的数据页类型有：索引页、数据页、undo页、插入缓冲、自适应哈希索引、锁信息、数据字典信息等。其中前4类会持久化存储。

对于数据的读操作，会先判断数据页是否在缓冲池中，如果在称之为缓存命中，直接读取数据页，如果不在，则读取磁盘上的页，同时将磁盘中读取的页存放在缓冲池中。对于数据的写操作，则首先修改缓冲池中的页，然后再以一定的频率刷新到磁盘上(Checkpoint 机制)。

缓冲池的大小直接影响了数据库的性能。多线程对 buffer pool 进行访问的时候会发生资源竞争，Mysql 通过支持buffer pool进行sharding来提高并发性能，涉及参数为innodb\_buffer\_pool\_size、innodb\_buffer\_pool\_instances。每个instance的size = innodb\_buffer\_pool\_size/innodb\_buffer\_pool\_instances

官方推荐sharding后单个instance 的size不要小于 1GB。

1. mysql> show variables like 'innodb\_buffer\_pool\_size'; //缓冲池大小
2. +-------------------------+-----------+
3. | Variable\_name           | Value     |
4. +-------------------------+-----------+
5. | innodb\_buffer\_pool\_size | 134217728 |
6. +-------------------------+-----------+
7. mysql> show variables like 'innodb\_buffer\_pool\_instances'; //缓冲池实例数量
8. +------------------------------+-------+
9. | Variable\_name                | Value |
10. +------------------------------+-------+
11. | innodb\_buffer\_pool\_instances | 1     |
12. +------------------------------+-------+

MySQL 5.6 版本开始，可以通过表information\_schema.innodb\_buffer\_pool\_stats 查看缓冲状态。

在InnoDB 引擎中，缓冲池中页的大小默认为16KB，缓冲池是通过略有改进的LRU算法来进行管理的，涉及LRU List、Free List、Flush List三个存储单元。对于新读取到的页，并不是直接放到LRU列表首部，而是放到列表中称之为mindpoint 的位置，该位置由参数innodb\_old\_blocks\_pct 控制。

1. mysql> show variables like '%innodb\_old\_blocks\_pct%';
2. +-----------------------+-------+
3. | Variable\_name         | Value |
4. +-----------------------+-------+
5. | innodb\_old\_blocks\_pct | 37    |  //新读取的页插入到LRU列表的37%位置。
6. +-----------------------+-------+
7. mysql> show variables like '%innodb\_old\_blocks\_time%';
8. +------------------------+-------+
9. | Variable\_name | Value |
10. +------------------------+-------+
11. | innodb\_old\_blocks\_time | 1000 | //新读取的页在midpoint位置停留1000微秒后移入new 列表中
12. +------------------------+-------+

mindpoint 之后的列表称之为old List，之前的为new List，new List中存储的页是最为活跃的热点数据。

初始化时LRU列表是空的，所有可用的空闲页存放在Free List中，需要从缓冲池中分配页时，查询Free List中是否有可用页，若有则放入LRU List中，同时从Free List删除。否则淘汰LRU列表末尾的页，将对应的内存空间分配给新页。LRU List 中被修改的页成为脏页(dirty page)，表现为缓冲池中的页和磁盘上的页数据不一致，这时会通过checkpoint 机制将脏页刷回磁盘，脏页存储在Flush List 中。

1. mysql> show engine innodb status \G
2. ----------------------
3. BUFFER POOL AND MEMORY
4. ----------------------
5. Total large memory allocated 137428992
6. Dictionary memory allocated 340675
7. Buffer pool **size**   8191  // 缓冲池总大小
8. **Free** buffers       7431  // **Free** List中页的数量
9. **Database** pages     759   // LRU List中页的数量
10. Old **database** pages 270   // old List 中页的数量
11. Modified db pages  0     // 脏页数量
12. Pending reads      0
13. Pending writes: LRU 0, flush list 0, single page 0
14. Pages made young 0, not young 0  //LRU 列表中页移动到new端的数量
15. 0.00 youngs/s, 0.00 non-youngs/s //每秒的操作次数
16. Pages **read** 633, created 126, written 717
17. 0.00 reads/s, 0.00 creates/s, 0.00 writes/s
18. **No** buffer pool page gets since the **last** printout
19. Pages **read** ahead 0.00/s, evicted without access 0.00/s, Random **read** ahead 0.00/s
20. LRU len: 759, unzip\_LRU len: 0 //16Kb的页由 LRU 管理，非16Kb的由 unzip\_LRU 管理
21. I/O sum[0]:cur[0], unzip sum[0]:cur[0]

##### 插入缓冲

Insert Buffer用于提升非聚集索引的插入性能。它的使用需要满足下面条件：

* 索引是辅助索引；
* 索引不是唯一的；要求唯一的话，需要额外的查找，会导致离散读取情况。

对于非聚集索引的插入或者更新操作，先判断要插入的索引页是否在缓冲池中，若在则直接插入，若不在则先放入insert buffer对象中，然后进行insert buffer和辅助索引页子节点的合并操作，通常可以将多个插入合并到一个操作中。类比图书馆还书，管理员放回书架的操作。

涉及参数，bulk\_insert\_buffer\_size。

1. mysql> show variables like '%buf%';
2. +-------------------------------------+----------------+
3. | Variable\_name                       | Value          |
4. +-------------------------------------+----------------+
5. | bulk\_insert\_buffer\_size             | 8388608        |  //默认 8MB
6. +-------------------------------------+----------------+

##### Change Buffer

Change Buffer 是 Mysql 5.5 版本之后对insert buffer 的升级。Change Buffer 支持对DML操作insert、delete、update的缓冲，分别对应insert buffer、delete buffer、purge buffer。通过参数innodb\_change\_buffering参数来开启或者关闭各种buffer。

1. +-------------------------------+-------+
2. | Variable\_name                 | Value |
3. +-------------------------------+-------+
4. | innodb\_change\_buffer\_max\_size | 25    |  //最大内存使用百分比
5. | innodb\_change\_buffering       | all   |   //inserts、deletes、purges、changes、all、none。changes对应开启inserts和deletes。
6. +-------------------------------+-------+

查看引擎buffer信息。

1. mysql> show engine innodb status \G
2. -------------------------------------
3. **INSERT** BUFFER AND ADAPTIVE HASH **INDEX**
4. -------------------------------------
5. Ibuf: **size** 1, **free** list len 0, seg **size** 2, 0 merges
6. merged operations:
7. **insert** 0, **delete** mark 0, **delete** 0
8. discarded operations:
9. **insert** 0, **delete** mark 0, **delete** 0
10. Hash **table** **size** 34673, node heap has 0 buffer(s)
11. Hash **table** **size** 34673, node heap has 0 buffer(s)
12. Hash **table** **size** 34673, node heap has 0 buffer(s)
13. Hash **table** **size** 34673, node heap has 0 buffer(s)
14. Hash **table** **size** 34673, node heap has 1 buffer(s)
15. Hash **table** **size** 34673, node heap has 0 buffer(s)
16. Hash **table** **size** 34673, node heap has 0 buffer(s)
17. Hash **table** **size** 34673, node heap has 0 buffer(s)
18. 0.00 hash searches/s, 0.00 non-hash searches/s

Change buffer 只限于用在普通索引的场景下，而不适用于唯一索引。那么，现在有一个问题就是：普通索引的所有场景，使用 change buffer 都可以起到加速作用吗？因为 merge 的时候是真正进行数据更新的时刻，而 change buffer 的主要目的就是将记录的变更动作缓存下来，所以在一个数据页做 merge 之前，change buffer 记录的变更越多（也就是这个页面上要更新的次数越多），收益就越大。因此，对于**写多读少的业务来说，页面在写完以后马上被访问到的概率比较小，此时 change buffer 的使用效果最好**。这种业务模型常见的就是账单类、日志类的系统。反过来，假设一个业务的更新模式是写入之后马上会做查询，那么即使满足了条件，将更新先记录在 change buffer，但之后由于马上要访问这个数据页，会立即触发 merge 过程。这样随机访问 IO 的次数不会减少，反而增加了 change buffer 的维护代价。所以，对于这种业务模式来说，change buffer 反而起到了副作用。

对于唯一索引来说，所有的更新操作都要先判断这个操作是否违反唯一性约束。比如，要插入 (4,400) 这个记录，就要先判断现在表中是否已经存在 k=4 的记录，而这必须要将数据页读入内存才能判断。如果都已经读入到内存了，那直接更新内存会更快，就没必要使用 change buffer 了。因此，唯一索引的更新就不能使用 change buffer。

##### 两次写

InnoDB的Page Size一般是16KB，其数据校验也是针对这16KB来计算的，将数据写入到磁盘是以Page为单位进行操作的。由于文件系统对数据页大多数情况下不是原子操作，这意味着如果服务器宕机了，可能只完成了部分写入，其余部分写入失败，这种情况下就是partial page write问题。如果想通过redo log 恢复数据必须有完整的页数据，也就是待恢复数据所在的页必须是完整的，如果页已经损坏，是无法重做的。因为MySQL在恢复的过程中需要检查page的checksum，发生partial page write问题时，page已经损坏从而无法通过checksum验证，就无法恢复。

InnoDB 通过两次写（double write）来解决这个问题。其工作流程是如下：

1. 触发数据缓冲池中的脏页进行刷新到data file的时候，并不直接写磁盘，而是会通过memcpy函数将脏页先复制到内存中的double write buffer，之后通过double write buffer再分两次、每次1MB顺序写入共享表空间的物理磁盘上，然后马上调用fsync函数，同步脏页进磁盘上。由于在这个过程中，double write页的存储时连续的，因此写入磁盘为顺序写，性能很高；
2. 完成double write后，再将脏页写入实际的各个表空间文件，这时写入就是离散写。
3. 数据在写入过程崩溃时，首先从共享表空间中获取该页的副本，将其拷贝到表空间文件，然后应用redo log 进行数据恢复。

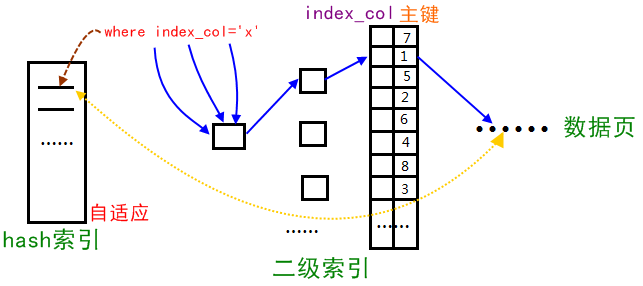
参数innodb\_doublewrite 控制是否开启两次写功能，建议一致开启该功能。

1. mysql> show variables like '%doublewrite%';
2. +--------------------+-------+
3. | Variable\_name      | Value |
4. +--------------------+-------+
5. | innodb\_doublewrite | **ON**    |
6. +--------------------+-------+

如果两次写页本身发生问题，则数据页面不会有写操作，此时系统故障，有问题的是两次写页，对应的数据页还在buffer里面，文件中的值还是之前事务操作的结果，还是一致状态。

##### 自适应哈希索引

InnoDB 会根据某些页的访问频率和模式建立索引，提高读写性能。



Innodb存储引擎通过监控表上二级索引的访问频率，如果发现某二级索引被频繁访问，该二级索引数据会自动被生成到hash索引里面去，自适应哈希索引通过缓冲池的B+树构造而来，因此建立的速度很快。

自适应hash索引只适合搜索等值的查询，如select \* from table\_t where index\_col='xxx'，而对于其他查找类型，如范围查找，是不能使用的。

涉及参数innodb\_adaptive\_hash\_index。

1. mysql> show variables like 'innodb\_adaptive\_hash\_index';
2. +----------------------------------+-------+
3. | Variable\_name                    | Value |
4. +----------------------------------+-------+
5. | innodb\_adaptive\_hash\_index       | **ON**    |  // 默认开启
6. | innodb\_adaptive\_hash\_index\_parts | 8     |  // 索引被切成8个分区
7. +----------------------------------+-------+

##### 刷新临接页

当刷新一个脏页的时候，引擎会检测该页所在的区的所有页，如果是脏页则会一同刷新。设计参数 innodb\_flush\_neighbors。

1. mysql> show variables like 'innodb\_flush\_neighbors';
2. +------------------------+-------+
3. | Variable\_name          | Value |
4. +------------------------+-------+
5. | innodb\_flush\_neighbors | 1     |  //默认开启
6. +------------------------+-------+

##### 引擎开启与关闭

主要关注两个参数 innodb\_fast\_shutdown与innodb\_force\_recover。

1. mysql> show variables like 'innodb\_fast\_shutdown';
2. +----------------------+-------+
3. | Variable\_name        | Value |
4. +----------------------+-------+
5. | innodb\_fast\_shutdown | 1     |  //可取0 ，1，2
6. 0：innodb 需要完成所有的full purge和merge insert buffer
7. 1：不需要完成0条件下所有的上述操作，但需要刷新缓冲池中的脏页数据
8. 2：不需要完成1条件下的操作，仅把日志写入日志文件，在下次启动时，需要恢复数据
9. +----------------------+-------+
10. mysql> show variables like 'innodb\_force\_recovery';
11. +-----------------------+-------+
12. | Variable\_name         | Value |
13. +-----------------------+-------+
14. | innodb\_force\_recovery | 0     |  //可取0，1，2，3，4，5，6
15. +-----------------------+-------+

##### 内存堆

维护引擎进程、线程运行时的数据结构，诸如LRU、锁、数据表、库原数据等信息。

##### 重做日志缓冲

重做日志缓冲(redo log buffer)，负责存储重做日志信息，然后按照一定的频率将其刷新到重做日志文件，其大小由参数innodb\_log\_buffer\_size 控制，

1. mysql> show variables like 'innodb\_log\_buffer\_size';
2. +------------------------+----------+
3. | Variable\_name          | Value    |
4. +------------------------+----------+
5. | innodb\_log\_buffer\_size | 16777216 |  //默认为16MB
6. +------------------------+----------+

该缓冲一般不需要设置太大，在下列三种情况会将重做日志缓冲中的内容刷新到磁盘上的重做日志文件中。

* Master Thread 每一秒将重做日志缓冲刷新到重做日志文件；
* 每个事物提交时，会将重做日志缓冲刷新到重做日志文件；
* 重做日志缓冲剩余空间小于1/2时，重做日志缓冲刷新到重做日志文件；

##### Master Thread

负责将缓冲池中数据异步刷新到磁盘，保证数据一致性，包括脏页的刷新、合并插入缓冲、UNDO 页的回收等。

Master Thread 具有最高优先级。其内部包括主循环、后台循环、刷新循环、暂停循环，它会根据数据库运行状态在上述循环中切换。实现逻辑伪代码：

1. void master\_thread(){
2. **goto** loop;
3. loop：
4. **for**(**int** i = 0; i<10; i++){
5. thread\_sleep(1) // sleep 1 **second**
6. do log buffer flush **to** disk
7. if ( last\_one\_second\_ios < 5)   **//过去1秒内io数小于5**
8. do merge at most 5 **insert** buffer
9. if ( buf\_get\_modified\_ratio\_pct > innodb\_max\_dirty\_pages\_pct )**// 缓冲池中脏页比例**
10. do buffer pool flush 100%  dirty page   // **剥离到 page cleaner thread**
11. **else** if enable adaptive flush
12. do buffer pool flush desired amount dirty page
13. if ( **no** user activity )
14. **goto** backgroud loop
15. }
16. if ( last\_ten\_second\_ios < innodb\_io\_capacity)   **//过去10秒内io数**
17. do buffer pool flush 100%  dirty page
18. do merge **at** most 5% innodb\_io\_capacity **insert** buffer  //合并插入缓冲
19. do log buffer flush **to** disk    **//刷新重做日志缓冲**
20. do **full** purge  **//删除无用undo 页码，默认为20个**，
21. if ( buf\_get\_modified\_ratio\_pct > 70% )
22. do buffer pool flush 100%  dirty page
23. **else**
24. do buffer pool flush 10%  dirty page
25. do fuzzy **checkpoint**
26. **goto** loop
27. background loop:
28. do **full** purge    **// 剥离到 purge thread**
29. do merge 100%  **insert** buffer
30. if not idle:
31. **goto** loop:
32. **else**:
33. **goto** flush loop
34. flush loop:
35. do buffer pool flush 100%  dirty page
36. if ( buf\_get\_modified\_ratio\_pct> innodb\_max\_dirty\_pages\_pct )
37. go **to** flush loop
38. **goto** suspend loop
39. suspend loop:
40. suspend\_thread()
41. waiting event
42. **goto** loop;
43. }

上述代码涉及变量innodb\_io\_capacity、innodb\_max\_dirty\_pages\_pct。前者用于控制IO操作数量，当使用SSD磁盘或者做了RAID时候，可以将其值调高。后者控制默认的脏页比例。

1. mysql> show variables like 'innodb\_io\_capacity';
2. +--------------------+-------+
3. | Variable\_name      | Value |
4. +--------------------+-------+
5. | innodb\_io\_capacity | 200   |
6. +--------------------+-------+
7. mysql> show variables like 'innodb\_max\_dirty\_pages\_pct';
8. +----------------------------+-----------+
9. | Variable\_name              | Value     |
10. +----------------------------+-----------+
11. | innodb\_max\_dirty\_pages\_pct | 75.000000 |

##### IO Thread

负责AIO请求的回调处理。InnoDB中使用了大量的AIO（Async IO）操作来处理IO 请求，以提高数据库的性能。当前版本的引擎有4类IO Thread，write、read、insert buffer、log thread。

1. --------
2. FILE I/O
3. --------
4. I/O thread 0 state: waiting **for** completed aio requests (**insert** buffer thread)
5. I/O thread 1 state: waiting **for** completed aio requests (log thread)
6. I/O thread 2 state: waiting **for** completed aio requests (**read** thread)
7. I/O thread 3 state: waiting **for** completed aio requests (**read** thread)
8. I/O thread 4 state: waiting **for** completed aio requests (**read** thread)
9. I/O thread 5 state: waiting **for** completed aio requests (**read** thread)
10. I/O thread 6 state: waiting **for** completed aio requests (write thread)
11. I/O thread 7 state: waiting **for** completed aio requests (write thread)
12. I/O thread 8 state: waiting **for** completed aio requests (write thread)
13. I/O thread 9 state: waiting **for** completed aio requests (write thread)
14. show variables like 'innodb\_%io\_threads';
15. +-------------------------+-------+
16. | Variable\_name           | Value |
17. +-------------------------+-------+
18. | innodb\_read\_io\_threads  | 4     |   // **read**
19. | innodb\_write\_io\_threads | 4     |   // write
20. +-------------------------+-------+

##### Purge Thread

用于回收不需要的undo页，清理page里面的有“deleted”标签的数据行。

1. mysql> show variables like 'innodb\_purge\_threads';
2. +----------------------+-------+
3. | Variable\_name        | Value |
4. +----------------------+-------+
5. | innodb\_purge\_threads | 4     |
6. +----------------------+-------+

##### Page Cleaner Thread

用于刷新脏页数据。

1. mysql> show variables like 'innodb\_%\_cleaners';
2. +----------------------+-------+
3. | Variable\_name        | Value |
4. +----------------------+-------+
5. | innodb\_page\_cleaners | 1     |   //默认为1，最大值是64
6. +----------------------+-------+

##### Checkpoint

checkpoint 目的是解决以下问题：

* 缩短数据库恢复时间；数据库宕机宕机时，不需要重做所有的日志，checkpoint 之前的页都已经刷回到磁盘，只需对checkpoint后的重做日志进行恢复。
* 缓冲池不够用时，将脏页刷新到磁盘；缓冲池不够用时，LRU会淘汰最近最少使用的页，这些页若为脏页，需要强制执行checkpoint，将脏页刷回磁盘。
* 重做日志不可用时，刷新脏页；重做日志再逻辑上是一个环形列表，重做日志可重用是指即当数据库发生宕机时，数据库恢复操作不需要这部分日志，因此可以被覆盖。如若这部分日志还需要使用，那必须强制产生checkpoint，将缓冲池中的页刷新到当前重做日志的位置。

InnoDB引擎有两类checkpoint，Sharp checkpoint以及Fuzzy checkpoint。

Sharp checkpoint 在数据库关闭时将所有脏页都刷回到磁盘，该行为由参数innodb\_fast\_shutdown（0/1为正常关闭）控制。

Fuzzy checkpoint在数据库运行时，刷新部分脏页到磁盘。可能的checkpoint 如下：

* Master Thread Checkpoint, 以固定的频率(每秒)刷新一定比例的脏页到磁盘；
* Flush LRU List Checkpoint, LRU 淘汰某些页时，若果涉及脏页需要刷回到磁盘；操作由page cleaner thread 完成。
* Async/Sync Flush Checkpoint，重做日志不可用时，需要刷新某些页到磁盘，保证重做日志循环可用性。操作由page cleaner thread 完成。
* Dirty Page Too Much checkpoint，脏页比例太多，强制刷回到磁盘，该比例由参数 innodb\_max\_dirty\_pages\_pct控制。

#### 文件系统

Mysql 数据库中，常用的文件包括参数文件、日志文件、表结构文件、存储引擎文件。

##### 日志文件

常用的日志文件有错误日志（error log）、二进制日志 （binlog）、查询日志(log)以及慢查询日志(slow query log)。

###### 错误日志

Mysql 在启动、运行以及关闭过程出现的异常均记录在错误日志中。该文件不仅记录了所有的错误信息，也记录了某些警告信息。涉及参数log\_error。

1. mysql> show variables like 'log\_error';
2. +---------------+--------------------------+
3. | Variable\_name | Value                    |
4. +---------------+--------------------------+
5. | log\_error     | /var/log/mysql/error.log |  //文件路径
6. +---------------+--------------------------+

###### 二进制日志

二进制日志(binary log)记录了所有对数据库执行更改的操作。该日志主要用于：

* 恢复数据
* 复制数据
* 审计

与二进制日志相关的参数

1. mysql> show variables like 'log\_bin';
2. +---------------+-------+
3. | Variable\_name | Value |
4. +---------------+-------+
5. | log\_bin       | **ON**    |  // 开启二进制日志
6. +---------------+-------+
7. mysql> show variables like 'datadir';
8. +---------------+-----------------+
9. | Variable\_name | Value           |
10. +---------------+-----------------+
11. | datadir       | /var/lib/mysql/ |   // 二进制日志文件路径
12. +---------------+-----------------+
13. mysql> system sudo ls -lh /var/lib/mysql/
14. bin\_log.000001 //二进制日志
15. bin\_log.index //二进制的索引文件，存储二进制日志文件列表
16. mysql> show variables like '%binlog%';
17. +--------------------------------------------+----------------------+
18. | Variable\_name                              | Value                |
19. +--------------------------------------------+----------------------+
20. | binlog\_cache\_size                          | 32768                |
21. 指定待提交二进制日志缓冲区大小，默认为32k，该参数是基于会话的，每个会话线程独占一个。该值得设定参考参数Binlog\_cache\_disk\_use(使用临时文件写二进制的次数,越小越好)和Binlog\_cache\_use(使用缓冲写二进制的次数)。
22. | binlog\_checksum                            | CRC32                |
23. | binlog\_direct\_non\_transactional\_updates    | **OFF**                  |
24. | binlog\_error\_action                        | ABORT\_SERVER         |
25. | binlog\_format                              | ROW                  |
26. 指定二进制日志格式。默认为ROW。可选值为STATEMENT，MIXED和ROW。
27. | binlog\_group\_commit\_sync\_delay             | 0                    |
28. | binlog\_group\_commit\_sync\_no\_delay\_count    | 0                    |
29. | binlog\_gtid\_simple\_recovery                | **ON**                   |
30. | binlog\_max\_flush\_queue\_time                | 0                    |
31. | binlog\_order\_commits                       | **ON**                   |
32. | binlog\_row\_image                           | **FULL**                 |
33. | binlog\_rows\_query\_log\_events               | **OFF**                  |
34. | binlog\_stmt\_cache\_size                     | 32768                |
35. | binlog\_transaction\_dependency\_history\_size | 25000                |
36. | binlog\_transaction\_dependency\_tracking     | COMMIT\_ORDER         |
37. | innodb\_api\_enable\_binlog                   | **OFF**                  |
38. | innodb\_locks\_unsafe\_for\_binlog             | **OFF**                  |
39. | log\_statements\_unsafe\_for\_binlog           | **ON**                   |
40. | max\_binlog\_cache\_size                      | 18446744073709547520 |
41. | max\_binlog\_size                            | 104857600            |
42. 指定单个二进制文件的最大值，默认1MB。超过改值则产生基于前面文件后缀名+1的新的文件,并记录到.index文件。
43. | max\_binlog\_stmt\_cache\_size                 | 18446744073709547520 |
44. | sync\_binlog                                | 1                    |
45. 指定二进制日志同步磁盘的行为。默认为1。0表示不控制binlog的刷新，由系统去操作，默认每秒1次，此时性能最高。1 表示每次事务执行都会在写缓冲时数据同步写盘，此时数据最安全。N(>1) 表示每写缓冲多少次就同步到磁盘。
46. +--------------------------------------------+----------------------+

二进制日志文件查看工具，mysqlbinlog。

###### 查询日志

查询日志记录了所有对数据库请求的信息，包括正确执行、未正确执行的SQL语句。默认文件名为$hostname.log。该文件可以通过vim等工具直接查看。

###### 慢查询日志

定位问题SQL，优化性能的最佳线索。90%的数据库性能问题都是由于SQL语句写的不当造成的。

1. mysql> show variables like '%slow\_query%';
2. +---------------------+----------------------------------------+
3. | Variable\_name       | Value                                  |
4. +---------------------+----------------------------------------+
5. | slow\_query\_log      | **OFF**                                    |
6. //默认未开启 慢查询日志，建议设置为ON
7. | slow\_query\_log\_file | /var/lib/mysql/VM-0-14-ubuntu-slow.log |
8. // 慢查询日志文件路径
9. +---------------------+----------------------------------------+
10. mysql> show variables like 'long\_query\_time';
11. +-----------------+-----------+
12. | Variable\_name   | Value     |
13. +-----------------+-----------+
14. | long\_query\_time | 10.000000 |  //sql 语句执行时间阈值，默认为10秒。
15. +-----------------+-----------+
16. mysql> show variables like 'log\_queries\_not\_using\_indexes';
17. +-------------------------------+-------+
18. | Variable\_name                 | Value |
19. +-------------------------------+-------+
20. | log\_queries\_not\_using\_indexes | **OFF**   |  //设置为ON的情况下，运行的SQL没有使用索引且耗时超过阈值，也会记录到慢查询日志。
21. +-------------------------------+-------+
22. mysql> show variables like 'log\_throttle%';
23. +----------------------------------------+-------+
24. | Variable\_name                          | Value |
25. +----------------------------------------+-------+
26. | log\_throttle\_queries\_not\_using\_indexes | 0     |  //每分钟允许记录到慢查询日志且未使用索引的SQL语句次数,默认为0,表示无限制。
27. +----------------------------------------+-------+
28. mysql> show variables like 'min\_examined\_row\_limit';
29. +------------------------+-------+
30. | Variable\_name          | Value |
31. +------------------------+-------+
32. | min\_examined\_row\_limit | 0     |  //对于查询扫描行数小于此参数的SQL，将不会记录到慢查询日志中；
33. +------------------------+-------+

慢查询日志分析工具 mysqldumlslow。

当前版本也可以通过表mysql.slow\_log查询慢查询记录。前提是将参数log\_output设定为TABLE。

1. mysql> show variables like 'log\_output';
2. +---------------+-------+
3. | Variable\_name | Value |
4. +---------------+-------+
5. | log\_output    | **TABLE** |
6. +---------------+-------+

##### 表结构文件

Mysql 数据的存储是依附于表的，每个表都有与之对应的文件，以后缀.frm命名，文件记录了对应表的表结构定义。同时该文件还用于存储视图的定义。

##### InnoDB 存储引擎文件

存储引擎文件包括表空间文件、重做日志文件、undo log。

###### 表空间文件

InnoDB 将数据按表空间进行存放。默认的表空间文件是ibdata1，所有表的数据都会记录到该共享表空间。

1. mysql> system sudo ls -lh /var/lib/mysql
2. -rw-r----- 1 mysql mysql  76M Jan 15 19:07 ibdata1 // 表空间文件
3. mysql> show variables like 'innodb\_data\_file\_path';
4. +-----------------------+------------------------+
5. | Variable\_name         | Value                  |
6. +-----------------------+------------------------+
7. | innodb\_data\_file\_path | ibdata1:12M:autoextend |  //默认12MB，自动增长
8. +-----------------------+------------------------+

如果设置了参数innodb\_file\_per\_table 则每个表对应一个独立的表空间，表名.ibd。独立表空间存储表数据、索引、插入缓冲等信息，其余数据还是存放于共享表空间。

1. mysql> show variables like 'innodb\_file\_per\_table';
2. +-----------------------+-------+
3. | Variable\_name         | Value |
4. +-----------------------+-------+
5. | innodb\_file\_per\_table | **ON**    |
6. +-----------------------+-------+

###### 共享表空间

共享表空间ibdata1，存放内容包括：

1、数据字典(data dictionary)：记录数据库相关信息

2、doublewrite write buffer：解决部分写失败（页断裂）

3、insert buffer：内存insert buffer数据，周期写入共享表空间，防止意外宕机

4、回滚段(rollback segments)

5、undo空间：undo页

　影响共享表空间增长的对象：insert buffer、undo空间

1、什么时候undo暴涨：

　　1、大事务为主，例如修改了40万行才提交

　　2、长事务导致的undo持续增加

2、insert buffer空间很大

　　1、索引建立过多

　　2、很多索引不怎么使用

　　3、索引和主键顺序严重不一致：主键的建立选择有问题

所以ibdata1使用时要注意：在初始化ibdata1时，最好是设置大一些，避免由于高并发突然暴增，影响性能。初始化设置，后期设置就没有意义。

1、空间大小

　　innodb表是按表空间进行存放的，共享表空间ibdata1默认初始化大小是12M。

　　初始化：innodb\_data\_file\_path=ibdata1:10G:ibdata2:10G:autoextend

2、undo分离

　　undo最容易导致ibdata1暴涨，所以需要将undo分离出去，只能在初始化的时候做，初始化之后就没有办法实现undo分离。在初始化实例之前，只需要设置 innodb\_undo\_tablespaces 参数，默认是等于0，建议将该参数设置大于等于3，即可将undo log设置到单独的undo表空间中。

###### 重做日志文件

重做日志(redo log)，也叫日志文件,记录了存储引擎的事务日志,它是数据库管理系统与文件系统最核心的区别。用来对数据进行crash recovery，记录每个数据页的物理更改情况，减少page的刷盘量，将随机IO变为顺序IO。

存储引擎至少有一个重做日志文件组，文进组下至少有2个日志文件，默认重做日志文件是ib\_logfile0与ib\_logfile1。重做日志文件以循环写的方式运行。

1. mysql> show variables like 'innodb\_log\_group\_home\_dir';
2. +---------------------------+-------+
3. | Variable\_name             | Value |
4. +---------------------------+-------+
5. | innodb\_log\_group\_home\_dir | ./    |  //重做日志文件路径，默认是./，指数据库数据目录。
6. +---------------------------+-------+
8. mysql> system sudo ls -lh /var/lib/mysql
9. -rw-r----- 1 mysql mysql  48M Jan 15 19:07 ib\_logfile0
10. -rw-r----- 1 mysql mysql  48M Jun 11  2019 ib\_logfile1
11. mysql> show variables like 'innodb\_%log%';
12. +----------------------------------+------------+
13. | Variable\_name                    | Value      |
14. +----------------------------------+------------+         |
15. | innodb\_log\_file\_size             | 50331648   |  //
16. | innodb\_log\_files\_in\_group        | 2          |
17. | innodb\_log\_group\_home\_dir        | ./         |
18. +----------------------------------+------------+  |

重做日志首先写重做日志缓冲区（redo log buffer），然后顺序的写入日志文件(redo log file)。主线程不论事务是否提交，都会每秒执行一次写入操作，写入文件时是按照一个扇区的大小(512个字节)写入的。因为扇区（硬盘块大小是512字节）是最小的写入单位，可以保证写入时必定成功的，因此该过程不需要doublewrite。

MySQL支持用户自定义在commit时如何将log buffer中的日志刷log file中。这种控制通过变量 innodb\_flush\_log\_at\_trx\_commit 的值来决定。该变量可以取值：0、1、2，默认为1。

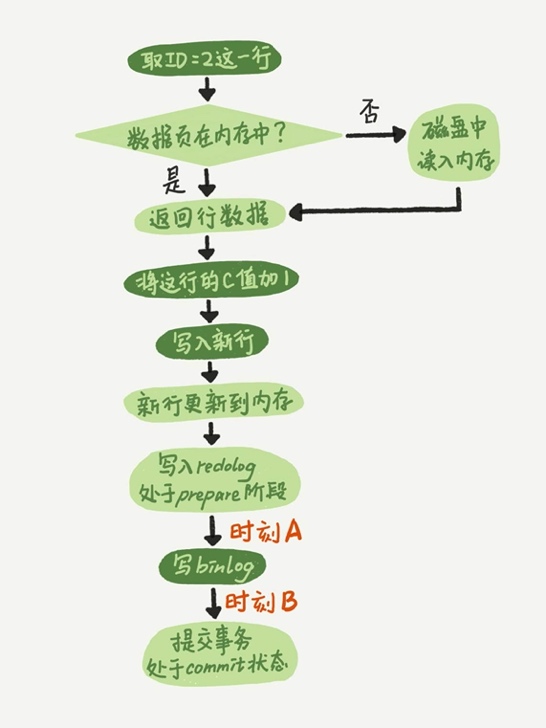
当设置为1的时候，事务每次提交都会将log buffer中的日志写入os buffer并调用fsync()刷到log file on disk中。这种方式即使系统崩溃也不会丢失任何数据，但是因为每次提交都写入磁盘，IO的性能较差。

当设置为0的时候，事务提交时不会将log buffer中日志写入到os buffer，而是每秒写入os buffer并调用fsync()写入到log file on disk中。也就是说设置为0时是(大约)每秒刷新写入到磁盘中的，当系统崩溃，会丢失1秒钟的数据。

当设置为2的时候，每次提交都仅写入到os buffer，然后是每秒调用fsync()将os buffer中的日志写入到log file on disk。

重做日志文件大小一般来说与buff pool size保证在10~5:1比较合适。

重做日志与二进制日志的两阶段提交。



在两阶段提交的不同时刻，MySQL 异常重启会出现什么现象。如果在图中时刻 A 的地方，也就是写入 redo log 处于 prepare 阶段之后、写 binlog 之前，发生了崩溃（crash），由于此时 binlog 还没写，redo log 也还没提交，所以崩溃恢复的时候，这个事务会回滚。这时候，binlog 还没写，所以也不会传到备库。时刻 B，也就是 binlog 写完，redo log 还没 commit 前发生 crash，那崩溃恢复的时候 MySQL 会怎么处理？如果 redo log 里面的事务只有完整的 prepare，则判断对应的事务 binlog 是否存在并完整：a. 如果是，则提交事务；b. 否则，回滚事务。

MySQL 怎么知道 binlog 是完整的?

一个事务的 binlog 是有完整格式的：statement 格式的 binlog，最后会有 COMMIT；row 格式的 binlog，最后会有一个 XID event。另外，在 MySQL 5.6.2 版本以后，还引入了 binlog-checksum 参数，用来验证 binlog 内容的正确性。对于 binlog 日志由于磁盘原因，可能会在日志中间出错的情况，MySQL 可以通过校验 checksum 的结果来发现。

redo log 和 binlog 是怎么关联起来的?

它们有一个共同的数据字段，叫 XID。崩溃恢复的时候，会按顺序扫描 redo log：如果碰到既有 prepare、又有 commit 的 redo log，就直接提交；如果碰到只有 parepare、而没有 commit 的 redo log，就拿着 XID 去 binlog 找对应的事务。

1. mysql> show engine innodb status \G
2. LOG
3. ---
4. Log sequence number 20670635
5. Log flushed up to   20670635
6. Pages flushed up to 20670635
7. Last checkpoint at  20670626
8. 0 pending log flushes, 0 pending chkp writes
9. 965 log i/o's done, 0.00 log i/o's/second

log sequence number就是当前的redo log(in buffer)中的lsn；

log flushed up to是刷到redo log file on disk中的lsn；

pages flushed up to是已经刷到磁盘的数据页上的lsn；

last checkpoint at是上一次检查点所在位置的lsn。

###### Undo log

Undo log只要用于提供回滚(rollback)和多版本并发控制(MVCC)。它保证了事务的原子性。默认存放在共享表空间中。

Undo log 是逻辑日志，可以认为当delete一条记录时，undo log中会记录一条对应的insert记录，反之亦然，当update一条记录时，它记录一条对应相反的update记录。

Undo log是采用段(segment)的方式来记录的，每个undo操作在记录的时候占用一个undo log segment。rollback segment称为回滚段，每个回滚段中有1024个undo log segment。另外，undo log也会产生redo log，因为undo log也要实现持久化存储。

1. mysql> show variables like '%undo%';
2. +--------------------------+------------+
3. | Variable\_name            | Value      |
4. +--------------------------+------------+
5. | innodb\_max\_undo\_log\_size | 1073741824 |
6. | innodb\_undo\_directory    | ./         |
7. | innodb\_undo\_log\_truncate | OFF        |
8. | innodb\_undo\_logs         | 128        |  //rollback segment 个数
9. | innodb\_undo\_tablespaces  | 0          |  //rollback segment 平均分配到多少个文件中，默认为0 表示全部写入到一个文件中。
10. +--------------------------+------------+

当事务提交的时候，InnoDB不会立即删除undo log，因为后续还可能会用到undo log，如隔离级别为repeatable read时，事务读取的都是开启事务时的最新提交行版本，只要该事务不结束，该行版本就不能删除，即undo log不能删除。但是在事务提交的时候，会将该事务对应的undo log放入到删除列表中，未来通过purge来删除。并且提交事务时，还会判断undo log分配的页是否可以重用，如果可以重用，则会分配给后面来的事务，避免为每个独立的事务分配独立的undo log页而浪费存储空间和性能。

通过undo log记录delete和update操作的结果发现：(insert操作无需分析，就是插入行而已)

delete操作实际上不会直接删除，而是将delete对象打上delete flag，标记为删除，最终的删除操作是purge线程完成的。

update分为两种情况：update的列是否是主键列。

如果不是主键列，在undo log中直接反向记录是如何update的。

如果是主键列，update分两部执行：先删除该行，再插入一行目标行。

#### 索引

##### B+树索引

B+树索引可分为聚集索引和辅助索引。聚集索引按照每张表的主键构造B+树，叶子节点存放的是整张表的行记录数据，称为数据页，数据页通过双向链表进行连接。聚集索引对于主键的排序查找和范围查找速度特别快。辅助索引（非聚集索引），叶子节点并不包含记录的全部数据。当通过辅助索引来查找数据的时候，引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键，然后通过主键索引找到完整的行记录。

1. mysql> show index **from** t \G
2. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
3. Table: t
4. Non\_unique: 0
5. Key\_name: PRIMARY
6. Seq\_in\_index: 1
7. Column\_name: id
8. Collation: A  //存储方式A或者NULL ,A 表示排序
9. Cardinality: 8  //表中唯一值数目的估计，analyze table 命令可以更新该值
10. Sub\_part: NULL  /列的部分索引
11. Packed: NULL  //关键字是否被压缩
12. Null:    //索引的列是否含有NULL
13. Index\_type: BTREE  //索引类型
14. Comment:
15. Index\_comment:

###### 联合索引

###### 覆盖索引

###### 优化器忽略索引

###### MMR

###### ICP

#### 锁

一致性锁定读

Lock in share mode;

For update

一致性非锁定读

READ COMMITTED、REPEATABLE READ。

自增锁

#### 事务

特性，ACID

D（持久性）：

重做日志用来实现事务的持久性。

LSN，日志序列号，8字节，可以表示：

重做日志写入总量

Checkpoint 位置

页的版本

### 性能优化

#### 池化技术

客户端和数据库服务器交互的一般流程是建立连接、操作数据、释放连接。如果在请求量大的情况下，频繁的建立和释放连接会带来严重的性能问题，这种情况可以通过池化技术进行优化。

在开发过程中我们会用到很多的连接池，像数据库连接池、HTTP 连接池、Redis 连接池等等。连接池的核心思想是预先建立若干连接，在使用的时候直接使用建立好的连接，使用完毕归还连接但并不释放。

数据库连接池有两个最重要的配置：最小连接数和最大连接数，它们控制着从连接池中获取连接的流程：

1. 如果当前连接数小于最小连接数，则创建新的连接处理数据库请求；
2. 如果连接池中有空闲连接则复用空闲连接；如果空闲池中没有连接并且当前连接数小于最大连接数，则创建新的连接处理请求；
3. 如果当前连接数已经大于等于最大连接数，则按照配置中设定的时间（C3P0 的连接池配置是 checkoutTimeout）等待旧的连接可用；
4. 如果等待超过了这个设定时间则向用户抛出错误。

连接池伪代码：

1. **class** ConnectionPool
2. {
3. **private**:
4. **int** c\_minSize;
5. **int** c\_curSize;
6. **int** c\_maxSize;
7. string c\_userName;
8. string c\_password;
9. string c\_hostPort;
10. **private**:
11. list<Connection\*> c\_List;
12. pthread\_mutex\_t c\_lock;
13. Driver \* c\_driver;
14. **friend** **class** auto\_ptr<ConnectionPool>;
15. **static** auto\_ptr<ConnectionPool> c\_ap\_instance;
16. **private**:
17. ConnectionPool(string c\_hostPort,string userName,string password,**int** maxSize);
18. ~ConnectionPool();
19. Connection \* CreateConnection(**void**);
20. **void** InitConnection(**int** initialize);
21. **void** TerminateConnection(Connection \* conn);
22. **void** TerminateConnectionPool(**void**);
23. **public**:
24. Connection \* GetConnection(**void**);
25. **void** ReleaseConnection(Connection \* conn);
26. **static** ConnectionPool &GetInstance(**void**);
27. };

完整代码参见 github。

连接池常见问题：

* 连接已建立，但不可用。这可能是由于数据库ip发生变化或者由于连接空闲时间超过参数wait\_timeout的阈值，数据库主动关闭连接，但这对于客户端是透明的。常用的解决手段是，启动一个线程来定期检测连接池中的连接是否可用，比如使用连接发送“select 1”的命令给数据库看是否会抛出异常，如果抛出异常则将这个连接从连接池中移除，并且尝试关闭。
* 连接数过大或者过小。对于数据库连接池，一般在线上建议最小连接数控制在 10 左右，最大连接数控制在 20～30 左右。当然实际的数量需要根据机器配置以及业务需求调整。
* 线程独占。当多个线程共用一个连接对象，而且各自都有自己的事务要处理时候，不能确定哪个数据库操作对应哪个事务，这是由于多个线程都在进行事务操作而引起的。为此我们可以使用每个线程独占一个连接来实现，虽然这种方法有点浪费连接池资源但是可以大大降低事务管理的复杂性。

#### 读写分离

读写分离的原理是将数据库读写操作分散到不同的节点上，其基本实现是：

1. 数据库服务器搭建主从集群，一主一从、一主多从都可以。
2. 数据库主机通过复制将数据同步到从机，每台从数据库服务器都存储了所有的业务数据。
3. 业务服务器将写操作发给数据库主机，将读操作发给数据库从机。主机负责读写操作，从机只负责读操作，这样即使写请求会锁表或者锁让记录也不会影响到读请求的执行。同时，在读流量比较大的情况下，可以部署多个从库共同承担读流量，提高系统性能。

读写分离的三个关键技术分别是主从复制、

MySQL 的主从复制依赖于 binlog 。其过程是：首先从库在连接到主节点时会创建一个 IO 线程，用以请求主库更新的 binlog，并且把接收到的 binlog 信息写入一个叫做 relay log 的日志文件中，而主库也会创建一个 log dump 线程来发送 binlog 给从库；同时，从库还会创建一个 SQL 线程读取 relay log 中的内容，并且在从库中做回放，最终实现主从的一致性。

##### 主从复制部署

GTID 搭建主从，摒弃使用二进制文件名称和日志位置的做法。

###### 一主多从

###### 主从互备

##### 主从复制关键问题

主从复制的设计的两个关键问题，主从复制延迟和主从访问路由。

##### 主从复制参数

server\_id: 设置mysql 实例的server\_id，每个实例的server\_id 必须唯一。

gtid\_mode=on: 开启GTID模式。

enforce\_gtid\_consistency=on:保证GTID的一致性。

log-bin:开启Binlog。

log-slave-updates=1:从机是否记录binlog。

binlog-format=ROW:建议设置为ROW格式。

##### 半同步复制

实战篇

#### 索引

重建索引

1. mysql> **create** **table** T(id **int** **primary** **key**, k **int** not null, **name** **varchar**(16),**index** (k))engine=InnoDB;
2. **alter** **table** T **drop** **index** k;
3. **alter** **table** T **add** **index**(k);
4. or
5. **alter** **table** T **drop** **primary** **key**;
6. **alter** **table** T **add** **primary** **key**(id);

重建索引 k 的做法是合理的，可以达到省空间的目的。但是，重建主键的过程不合理。不论是删除主键还是创建主键，都会将整个表重建。所以连着执行这两个语句的话，第一个语句就白做了。这两个语句，你可以用这个语句代替 ： alter table T engine=InnoDB。analyze table T 其实不是重建表，只是对表的索引信息做重新统计，没有修改数据，这个过程中加了 MDL 读锁；optimize table T 等于 recreate+analyze。

索引可能因为删除，或者页分裂等原因，导致数据页有空洞，重建索引的过程会创建一个新的索引，把数据按顺序插入，这样页面的利用率最高，也就是索引更紧凑、更省空间。

#### Join 操作

能不能使用 join 语句？

如果可以使用 Index Nested-Loop Join 算法，也就是说可以用上被驱动表上的索引，其实是没问题的；如果使用 Block Nested-Loop Join 算法，扫描行数就会过多。尤其是在大表上的 join 操作，这样可能要扫描被驱动表很多次，会占用大量的系统资源。所以这种 join 尽量不要用。所以你在判断要不要使用 join 语句时，就是看 explain 结果里面，Extra 字段里面有没有出现“Block Nested Loop”字样。

如果要使用 join，应该选择大表做驱动表还是选择小表做驱动表？

如果是 Index Nested-Loop Join 算法，应该选择小表做驱动表；如果是 Block Nested-Loop Join 算法：在 join\_buffer\_size 足够大的时候，是一样的；在 join\_buffer\_size 不够大的时候（这种情况更常见），应该选择小表做驱动表。所以，这个问题的结论就是，总是应该使用小表做驱动表。

在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，过滤完成之后，计算参与 join 的各个字段的总数据量，数据量小的那个表，就是“小表”，应该作为驱动表。

Join 语句可以怎么优化?

set optimizer\_switch='mrr=on,mrr\_cost\_based=off,batched\_key\_access=on';

BKA 优化是 MySQL 已经内置支持的，建议默认使用；

BNL 算法效率低，建议你都尽量转成 BKA 算法。优化的方向就是给被驱动表的关联字段加上索引；

基于临时表的改进方案，对于能够提前过滤出小数据的 join 语句来说，效果还是很好的；

MySQL 目前的版本还不支持 hash join，但你可以配合应用端自己模拟出来，理论上效果要好于临时表的方案。

#### Order by 原理

1. CREATE TABLE `t` (
2. `id` **int**(11) NOT NULL,
3. `city` varchar(16) NOT NULL,
4. `name` varchar(16) NOT NULL,
5. `age` **int**(11) NOT NULL,
6. `addr` varchar(128) DEFAULT NULL,
7. PRIMARY KEY (`id`),
8. KEY `city` (`city`)
9. ) ENGINE=InnoDB;
10. select city,name,age from t where city='杭州' order by name limit 1000 ;

##### 全字段排序

这个语句执行流程如下所示 ：

初始化 sort\_buffer，确定放入 name、city、age 这三个字段；

从索引 city 找到第一个满足 city='杭州’条件的主键 id，也就是图中的 ID\_X；

到主键 id 索引取出整行，取 name、city、age 三个字段的值，存入 sort\_buffer 中；

从索引 city 取下一个记录的主键 id；重复步骤 3、4 直到 city 的值不满足查询条件为止，对应的主键 id 也就是图中的 ID\_Y；

对 sort\_buffer 中的数据按照字段 name 做快速排序；

按照排序结果取前 1000 行返回给客户端。

##### raiwd排序

如果查询要返回的字段很多的话，那么 sort\_buffer 里面要放的字段数太多，这样内存里能够同时放下的行数很少，要分成很多个临时文件，排序的性能会很差。

SET max\_length\_for\_sort\_data = xxx;

max\_length\_for\_sort\_data，是 MySQL 中专门控制用于排序的行数据的长度的一个参数。它的意思是，如果单行的长度超过这个值，MySQL 就认为单行太大，要换一个算法。city、name、age 这三个字段的定义总长度是 36，我把 max\_length\_for\_sort\_data 设置为 16，我们再来看看计算过程有什么改变。新的算法放入 sort\_buffer 的字段，只有要排序的列（即 name 字段）和主键 id。

##### 全内存 or 使用外部文件

sort\_buffer\_size，就是 MySQL 为排序开辟的内存（sort\_buffer）的大小。如果要排序的数据量小于 sort\_buffer\_size，排序就在内存中完成。但如果排序数据量太大，内存放不下，则不得不利用磁盘临时文件辅助排序。

##### 不需要排序的 order by

并不是所有的 order by 语句，都需要排序操作的。MySQL 之所以需要生成临时表，并且在临时表上做排序操作，其原因是原来的数据都是无序的。如果对上述语句能够保证从 city 这个索引上取出来的行，天然就是按照 name 递增排序的话，就可以不用再排序了。可以在这个市民表上创建一个 city 和 name 的联合索引，对应的 SQL 语句是：

alter table t add index city\_user(city, name);

这样整个查询过程的流程就变成了：从索引 (city,name) 找到第一个满足 city='杭州’条件的主键 id；到主键 id 索引取出整行，取 name、city、age 三个字段的值，作为结果集的一部分直接返回；从索引 (city,name) 取下一个记录主键 id；重复步骤 2、3，直到查到第 1000 条记录，或者是不满足 city='杭州’条件时循环结束。

#### 临时表

临时表在使用上有以下几个特点：

* + 建表语法是 create temporary table …。
  + 一个临时表只能被创建它的 session 访问，对其他线程不可见。所以， session A 创建的临时表 t，对于 session B 就是不可见的。
  + 临时表可以与普通表同名。session A 内有同名的临时表和普通表的时候，show create 语句，以及增删改查语句访问的是临时表。show tables 命令不显示临时表。
  + 由于临时表只能被创建它的 session 访问，所以在这个 session 结束的时候，会自动删除临时表，建议养成用完后手动删除的习惯。
  + 在实现上，每个线程都维护了自己的临时表链表。这样每次 session 内操作表的时候，先遍历链表，检查是否有这个名字的临时表，如果有就优先操作临时表，如果没有再操作普通表；在 session 结束的时候，对链表里的每个临时表，执行 “DROP TEMPORARY TABLE + 表名”操作。
  + 在 binlog\_format='row’的时候，临时表的操作不记录到 binlog 中。
  + Union、group by 等操作会用到内存临时表。tmp\_table\_size 参数控制内存临时表的大小。

#### Union 执行过程

(select 1000 as f) union (select id from t1);

假设t表中有两行数据，id=1000和id=999;

这个语句的执行流程是这样的：

* 创建一个内存临时表，这个临时表只有一个整型字段 f，并且 f 是主键字段。
* 执行第一个子查询，得到 1000 这个值，并存入临时表中。
* 执行第二个子查询：拿到第一行 id=1000，试图插入临时表中。但由于 1000 这个值已经存在于临时表了，违反了唯一性约束，所以插入失败，然后继续执行；
* 取到第二行 id=999，插入临时表成功。
* 从临时表中按行取出数据，返回结果，并删除临时表，结果中包含两行数据分别是 1000 和 999。
* 如果把上面这个语句中的 union 改成 union all 的话，就没有了“去重”的语义。这样执行的时候，就依次执行子查询，得到的结果直接作为结果集的一部分，发给客户端。因此也就不需要临时表了。

#### Group by 执行过程

select id%10 as m, count(\*) as c from t1 group by m;

这个语句的执行流程是这样的：

* 创建内存临时表，表里有两个字段 m 和 c，主键是 m；
* 扫描表 t1 的索引 ，依次取出叶子节点上的 id 值，计算 id%10 的结果，记为 x；
* 如果临时表中没有主键为 x 的行，就插入一个记录 (x,1);
* 如果表中有主键为 x 的行，就将 x 这一行的 c 值加 1；
* 遍历完成后，再根据字段 m 做排序，得到结果集返回给客户端。

group by 的语义逻辑，是统计不同的值出现的个数。但是，由于每一行的 id%10 的结果是无序的，所以就需要有一个临时表，来记录并统计结果。那么，如果扫描过程中可以保证出现的数据是有序的可以省略掉临时表，提高性能。

在 MySQL 5.7 版本支持了 generated column 机制，用来实现列数据的关联更新。你可以用下面的方法创建一个列 z，然后在 z 列上创建一个索引（如果是 MySQL 5.6 及之前的版本，你也可以创建普通列和索引，来解决这个问题）。

alter table t1 add column z int generated always as(id % 10), add index(z);

索引 z 上的数据就是有序的了，上面的 group by 语句就可以改成：

select z, count(\*) as c from t1 group by z;

如果碰到不适合建索引的场景，还是需要排序的。

一个 group by 语句中需要放到临时表上的数据量特别大，是要按照“先放到内存临时表，插入一部分数据后，发现内存临时表不够用了再转成磁盘临时表”的逻辑执行。在 group by 语句中加入 SQL\_BIG\_RESULT 这个提示（hint），就可以告诉优化器：这个语句涉及的数据量很大，请直接用磁盘临时表。磁盘临时表是 B+ 树存储，存储效率不如数组来得高，所以优化器直会接用数组来存。因此，下面这个语句

select SQL\_BIG\_RESULT id%10 as m, count(\*) as c from t1 group by m;

的执行流程就是这样的：

初始化 sort\_buffer，确定放入一个整型字段，记为 m；

扫描表 t1 的索引 a，依次取出里面的 id 值, 将 id%10的值存入 sort\_buffer 中；

扫描完成后，对 sort\_buffer 的字段 m 做排序（如果 sort\_buffer 内存不够用，就会利用磁盘临时文件辅助排序）；

排序完成后，就得到了一个有序数组。根据有序数组，得到数组里面的不同值，以及每个值的出现次数。

#### Insert…select 语句

1. **CREATE** **TABLE** `t` (
2. `id` **int**(11) NOT NULL AUTO\_INCREMENT,
3. `c` **int**(11) **DEFAULT** NULL,
4. `d` **int**(11) **DEFAULT** NULL,
5. **PRIMARY** **KEY** (`id`),
6. **UNIQUE** **KEY** `c` (`c`)
7. ) ENGINE=InnoDB;
9. **insert** **into** t **values**(null, 1,1);
10. **insert** **into** t **values**(null, 2,2);
11. **insert** **into** t **values**(null, 3,3);
12. **insert** **into** t **values**(null, 4,4);
14. **create** **table** t2 like t ;

insert into t2(c,d) (select c+1, d from t force index(c) order by c desc limit 1);

这个语句的加锁范围，就是表 t 索引 c 上的 (3,4]和 (4,supremum]这两个 next-key lock，以及主键索引上 id=4 这一行。它的执行流程也比较简单，从表 t 中按照索引 c 倒序，扫描第一行，拿到结果写入到表 t2 中。

如果是下面的语句：

insert into t(c,d) (select c+1, d from t force index(c) order by c desc limit 1);

则执行过程为

创建临时表，表里有两个字段 c 和 d。

按照索引 c 扫描表 t，依次取 c=4、3、2、1，然后回表，读到 c 和 d 的值写入临时表。

这时，由于语义里面有 limit 1，所以只取了临时表的第一行，再插入到表 t 中。

也就是说，这个语句会导致在表 t 上做全表扫描，并且会给索引 c 上的所有间隙都加上共享的 next-key lock。所以，这个语句执行期间，其他事务不能在这个表上插入数据。至于这个语句的执行为什么需要临时表，原因是这类一边遍历数据，一边更新数据的情况，如果读出来的数据直接写回原表，就可能在遍历过程中，读到刚刚插入的记录，新插入的记录如果参与计算逻辑，就跟语义不符。由于实现上这个语句没有在子查询中就直接使用 limit 1，从而导致了这个语句的执行需要遍历整个表 t。它的优化方法也比较简单，就是用前面介绍的方法，先 insert into 到临时表 temp\_t，这样就只需要扫描一行；然后再从表 temp\_t 里面取出这行数据插入表 t1。使用内存临时表优化时，语句序列的写法如下：

1. **create** **temporary** **table** temp\_t(c **int**,d **int**) engine=memory;
2. **insert** **into** temp\_t  (**select** c+1, d **from** t **force** **index**(c) **order** **by** c **desc** limit 1);
3. **insert** **into** t **select** \* **from** temp\_t;
4. **drop** **table** temp\_t;

insert … select 是很常见的在两个表之间拷贝数据的方法。在可重复读隔离级别下，这个语句会给 select 的表里扫描到的记录和间隙加读锁。而如果 insert 和 select 的对象是同一个表，则有可能会造成循环写入。这种情况下，我们需要引入用户临时表来做优化。对于唯一索引，insert成功后，会加上X lock；对于后续涉及唯一索引的插入，需要在插入前进行duplicate key的检查，所以需要申请加上S lock，由于S lock与X lock不兼容，所以产生锁等待。因此，碰到由于唯一键约束导致报错后，要尽快提交或回滚事务，避免加锁时间过长。

看个具体例子

1. **CREATE** **TABLE** `uniq` (
2. `id` **int**(11) NOT NULL AUTO\_INCREMENT,
3. `aa` **int**(11) **DEFAULT** NULL,
4. `bb` **int**(11) **DEFAULT** NULL,
5. **PRIMARY** **KEY** (`id`),
6. **UNIQUE** **KEY** `uniq\_ab` (`aa`,`bb`)
7. ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8
8. session-1
9. **1**.mysql> **begin**;
10. Query OK, 0 **rows** affected (0.00 sec)
12. **2**. mysql> **insert** **into** uniq **values**(1,2,3);
13. Query OK, 1 row affected (0.00 sec)
15. **4**. mysql> **insert** **into** uniq **values**(2,2,4);
16. Query OK, 1 row affected (0.01 sec)
17. Session-2
18. 3**. insert** **into** uniq **values**(2,2,3);
19. **等待**
20. 执行4之后的结果
21. ERROR 1213 (40001): Deadlock found **when** trying **to** get lock; try restarting **transaction**

#### 再论rowid

主从配置下，数据库表没有主键，从库更新数据慢到卡死？

如果一个表没有主键，InnoDB会默认生成一个主键，但是主从同步依赖的是binlog，这个是server层面的东西，server层感知不到rawid的存在，复制的时候无法区分哪行对应哪行。因此MySQL使用的时候一定要有主键，如果没有主键，且处于ROW模式下，会出大事的。

###### Between and VS or VS in VS >= VS exist

#### SQL 语句优化小结

1. 不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

create table #t(...)

1. 很多时候用 exists 代替 in 是一个好的选择（a是大表，b是小表）：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

1. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2=100

应改为:select id from t where num=100\*2

1. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3)='abc'

select id from t where datediff(day,createdate,'2005-11-30')=0

应改为:

select id from t where name like 'abc%'

select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'

1. 下面的查询也将导致全表扫描：

select id from t where name like '%abc%'

对于 like '..%' (不以 % 开头)，可以应用 colunm上的index

1. 对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

1. 如果在 where 子句中使用参数，也会导致全表扫描。

因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时;它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num=@num

可以改为强制查询使用索引：

select id from t with(index(索引名)) where num=@num

1. 应尽量避免在 where 子句中使用 !=或<> 操作符，否则将引擎放弃使用索引而进行全表扫描。
2. 并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。
3. 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度;如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。
4. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。
5. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。
6. 绝对不要轻易用order by rand() ，很可能会导致mysql的灾难！
7. 每个表都应该设置一个ID主键，最好的是一个INT型，并且设置上自动增加的AUTO\_INCREMENT标志，这点其实应该作为设计表结构的第一件必然要做的事！！
8. 拆分大的 DELETE 或 INSERT 语句。因为这两个操作是会锁表的，表一锁住了，别的操作都进不来了，就我来说 有时候我宁愿用for循环来一个个执行这些操作。
9. 应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。
10. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。
11. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。
12. 任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

MySQL无法使用索引的情况总结

(1)字段使用函数，将无法使用索引

(2)Join 语句中 Join 条件字段类型不一致的时候 MySQL 无法使用索引

(3)复合索引的情况下，如果查询条件不包含索引列的最左边部分，即不满足最左前缀原则，则不会使用索引。

(4)如果mysql估计使用索引扫描比全表扫描更慢，则不使用索引。(扫描数据超过30%，都会走全表)

(5)以%开头的like查询

(6)数据类型出现隐式转换的时候也不会使用索引，特别是当列类型是字符串，那么一定记得在where条件中把字符串常量值用引号引起来，否则即便这个列上有索引，MySQL也不会用到，因为MySQL默认把输入的常量值进行转换以后才进行检索。

(7)用or分割开的条件，如果 or前的条件中的列有索引，而后面的列中没有索引，那么涉及的索引都不会被用到。

#### Explain 详解

1. mysql> explain **select** \* **from** t;
2. +----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------+
3. | id | select\_type | **table** | partitions | type | possible\_keys | **key**  | key\_len | ref  | **rows**   | filtered | Extra |
4. +----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------+
5. |  1 | SIMPLE      | t     | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 100318 |   100.00 | NULL  |
6. +----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------+

expain出来的信息有10列，分别是id、select\_type、table、type、possible\_keys、key、key\_len、ref、rows、Extra。

* id:选择标识符
* select\_type:表示查询的类型；
* table:输出结果集的表；
* partitions:匹配的分区；
* type:表示表的连接类型；
* possible\_keys:表示查询时，可能使用的索引；
* key:表示实际使用的索引；
* key\_len:索引字段的长度；
* ref:列与索引的比较；
* rows:扫描出的行数(估算的行数)；
* filtered:按表条件过滤的行百分比；
* Extra:执行情况的描述和说明；

这里仅强调select\_type和Extra 字段。

select\_type 示查询中每个select子句的类型。

1. SIMPLE(简单SELECT，不使用UNION或子查询等)；
2. PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部分，最外层的select被标记为PRIMARY)；
3. UNION(UNION中的第二个或后面的SELECT语句)；
4. DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)；
5. UNION RESULT(UNION的结果，union语句中第二个select开始后面所有select)；
6. SUBQUERY(子查询中的第一个SELECT，结果不依赖于外部查询)；
7. DEPENDENT SUBQUERY(子查询中的第一个SELECT，依赖于外部查询)；
8. DERIVED(派生表的SELECT, FROM子句的子查询)；
9. UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)；

Extra:

1. Using where， SQL使用了where条件过滤数据;
2. Using index， SQL所需要返回的所有列数据均在一棵索引树上，而无需访问实际的行记录。
3. Using index condition， SQL确实命中了索引，但不是所有的列数据都在索引树上，还需要访问实际的行记录。
4. Using filesort,SQL得到所需结果集，需要对所有记录进行文件排序。在一个没有建立索引的列上进行了order by，就会触发filesort，常见的优化方案是，在order by的列上添加索引，避免每次查询都全量排序。
5. Using temporary， SQL需要建立临时表(temporary table)来暂存中间结果。group by和order by同时存在，且作用于不同的字段时，就会建立临时表，以便计算出最终的结果集。
6. Using join buffer (Block Nested Loop)， SQL需要进行嵌套循环计算。常见的优化方案是，在关联字段上添加索引，避免每次嵌套循环计算。

### 扩展